A major purpose of the Technical Information Center is to provide the broadest dissemination possible of information contained in DOE's Research and Development Reports to business, industry, the academic community, and federal, state and local governments.

Although a small portion of this report is not reproducible, it is being made available to expedite the availability of information on the research discussed herein.

LA-UR -85-523

CONF-850266--1

Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36

TITLE: OPTIMIZING COMPUTATIONAL EFFICIENCY AND USER CONVENIENCE IN PLASMA SIMULATION CODES

AUTHOR(S): Christopher W. Barnes, X-1

LA-UR--85-523

DE85 007661

SUBMITTED TO: To be published in the proceedings of the Second International School for Space Simulations, Honolulu, Hawaii, February 3-16, 1985

DISCLAIMER

Los Alamos National Laboratory
Los Alamos, New Mexico 87545

OPTIMIZING COMPUTATIONAL EFFICIENCY AND USER CONVENIENCE IN PLASMA SIMULATION CODES

Christopher W. Barnes
Los Alamos National Laboratory
Los Alamos, New Mexico

ABSTRACT. It is usually important to write plasma simulation codes in such a way that they execute efficiently and are convenient to use. I discuss here practical techniques to achieve this goal. Numerical algorithms must be well formulated and advantage taken of machine architecture in casting the algorithm into a high level language such as Fortran. The advantages of writing critical routines in Assembler are discussed. For large simulation codes, disks must often be used as a temporary store for working data. Efficient methods for doing this are presented. Codes must not only be well organized for ease of implementation and maintenance, but also for ease of use. Ways are suggested for packaging codes such that setup, batch production, restarting and diagnostic postprocessing is facilitated. Particular emphasis is placed on graphics postprocessors, since they must be used in real time with graphics terminals as well as with hardcopy devices.

1. Introduction

The purpose of this tutorial lecture is to present what experience has shown to be good programming practice in the realization of plasma simulation codes for use on modern computers. I assume that you are well versed in a language such as fortran, and have used large computer installations in your everyday work. You may not yet have written and used a program that consists of many thousands of lines of source code and that consumes many hours of computer time. The implementation of non-trivial programs such as this requires careful planning and mindfulness of the physical nature of the computer system to be used, if efficiency and convenience are important. These considerations are likely to be important in a large code since the programming manpower needed is only justified by a long production life of the code.

It seems to be true that plasma simulation codes are never written once, and then used forever without modification. Instead, there is always new physics to include, new numerical algorithms to try or a change needed in the diagnostics package. You should assume that your new code will never be 'finished' and therefore should make ease of maintenance a primary consideration in its design.

Can we now identify the principal characteristics of plasma simulation codes that are relevant to optimizing efficiency and convenience? The most prominent features are the need for large amounts of computing time and the large amount of data used. The data base can exceed the storage capacity of the machine's central memory and might therefore need to be stored on disk and brought into memory as needed. The amount of computation done on a unit of data each time it is brought in from disk could be an issue, if it is desired to overlap disk transfers with computation. Particle codes are the worst offenders here. The particle move time can be as little as two microseconds on a CRAY-1 per timestep while the disk read/write time is several times this per particle. Simulation codes calculate the evolution of a plasma over time in a series of small timesteps, each step typically consisting of a particle or fluid element advance followed by an updating of the selfconsistent electric and magnetic fields. As the calculation proceeds, diagnostic data is produced in vast quantities and is either saved in raw or partly reduced form or is displayed or plotted directly. The diagnostic portions of a code can represent as much as 80 percent of the total number of sourcecode lines in the program. Fortunateley they usually can be made to use only a small part of the total compute time used in a run. They therefore need not be highly optimized.

Since these codes calculate the evolution of a plasma over time, it is desirable to be able to checkpoint the data occasionally so that a problem can be restarted without having to start from scratch. Often, a problem will be executed in several runs because of time constraints or to allow analysis of the problem results part way through. The ability to save the working data at a particular timestep and in a subsequent run restart from it is therefore considered essential in plasma simulation codes.

2. Optimization

In this section, I will discuss various ways to ensure that the code runs as efficiently as possible, without an excessive amount of programming effort. We will find that a rough knowledge of the computer architecture being used and some understanding of how the compiler makes use of that architectrue, will allow us to make substantial increases to execution efficiency.

2.1 A Vector Machine

The CRAY series of computers is now in wide spread use and is a prime example of what is known as a vector or pipeline machine. It has the ability to effectively perform an operation on a vector or array of data almost in parallel. Once the operation is initiated and after a short latency, the results appear one per machine cycle. The real power of this method lies in the fact that the overhead for index and address generation for the arithmetic or logical operation to be performed need only be done once, for the whole vector.

All vector machines also have the ability to do scalar operations, since some code is inherently unvectorizable. It is easy to show that vectorization can provide a factor of five speed increase over code executed only in the machine's scalar units. The objective then, when writing in Fortran for instance, is to convince the compiler to vectorize as much of the code as possible to achieve maximum speed. Even so, it is sometimes worth hand coding small portions of the program.

The maximum vector length allowable on the CRAY series of computers is 64 operands. For vectors longer than this, the compiler must break them into a series of vectors of length 64 or less. It is true that the hardware performs best when the vector length approaches 64. The breakeven point against scalar speed is a vector length of about 5.

A constraint against use of the vector units is that vector operands must be stored in memory with a constant 'stride', or spacing in address. This property makes writing of particle movers especially difficult, since the cell locations for a vector of particles are random, certainly not of constant 'stride'. It is worth noting that CRAY's upcoming X-MP48 will have the capability of doing 'scatter/gather' vector memory references, ie, non-constant stride. Bank conflicts will still degrade performance, however.

Memory is arranged in banks, or physical units such that as one steps through sequential memory addresses, a different bank is accessed each time until the number of banks has been cycled through, typically 16. Since the memory cycle time is much greater than a machine cycle, rapid hits to the same bank can slow access significantly. One should therefore avoid vector operations with strides which contain factors of 8 or 16 (or some similar power of two, depending on the number of banks in that particular machine).

2.2 Fortran Optimization

Optimization of Fortran codes is best done experimentally, since one can not always predict how the compiler will treat a given style of coding. Find a timing routine which reads a microsecond clock, or preferably a machine cycle clock. Then two calls to this routine will measure the time spent in the piece of code bracketing the two calls. The timing effect of coding changes can easily be seen using this method.

In scalar code, or code to be executed on a non-vector machine, anything can affect the timing. DO loops are a basic construct in Fortran. In any code unit containing DO loops, an innermost loop can be found where the majority of the time is spent. Unnecessary expressions should be kept out of the inner loop (sometimes the compiler will do this for you). Index calculations should be kept a simple as possible with the idea that each time through the loop, there will be as little calulation needed as possible. Sometimes non-changing portions of the index can be calculated outside the inner loop. Multi-dimensional arrays should be avoided. It is usually best to explicitely compute the mapping of a multi-dimensional index into a one-dimensional array. Use can be made of the fact that in Fortran, the first listed index is the fastest varying one, that is, is the contiguous one in storage.

In Fortran, vectorization is achieved not by special vector notation, but by finding suitable DO loops which lend themselves to vectorization. The array indices must not improperly overlap and the flow of the loop must not be interrupted. This means that there can be no IF tests within the loop. In a particle moving routine, the tests to see when a particle crosses a boundary will apparently not vectorize in Fortran, and should not be put in a loop with other operations which can. Interestingly, the machine instruction which performs the comparison has a vector form. The inclusion of a branch as a part of the IF construct makes it non-vectorizable.

When there are nested DO loops which are vectorizable, it is sometimes of advantage to reverse their order, although this is not always logically possible. I recently did this in a small FEL code in which the inner loop had a span of around 20 and the outer a span of over 200. The speedup under these conditions was significant although as the span of 20 was increased towards 64, the difference became negligible as one might expect.

I learned an interesting lesson about libraries in working on this code. At least half the time in the code was spent in calculating sine and cosine pairs (of the same argument). This was done using a call to a vectorized sine and then a vectorized cosine routine. I took the trouble to look at the CAL source code for these routines. I discovered that the same routine was called for both sine and cosine, and that because the vector arguments could be in any octant, that the routine always calculated both sine and cosine, and threw away the one not needed. I was able to modify the routine so that it returned both sine and cosine with one call, eliminating the need for the second call. Someone had thought of this before, of course. I found an entry point in the library to a COSSIN routine which did what I just described. Except it made calls to scalar routine which would have been far slower. Apparently the person who wrote and maintained the library had time only to make the COSSIN function look vectorizable to the calling routine, but not to actually provide the code needed. The lesson is apparently that library writers are human too.

2.3 Why Hand Code?

Some people are born hackers, and have written in Assembler for as long as they can remember. Others need some persuasion to learn and use assembly language in their codes. The truth is that it is often justified in codes such as particle codes, where most of the time is taken up by a few hundred lines of Fortran, the hand coding of which could achieve a speedup of a factor of two. Vector code is easy to hand code because you don't have to worry about the timing of the scalar part (indexing, counting, etc). In my view, vector code is the only kind I would consider worth the effort of hand coding.

The Cray machines have eight vector registers and a number of functional units each of which performs a specialized group of operations. They have names like "Floating Point Add Unit", "Floating Point Multiply Unit", "Shift Unit" and "Logical Function Unit". Memory references can be thought of as being performed by a special functional unit. In rough terms, the rules are that you can have as many functional units going at once as you want, so long as the rules of register conflict are observed. It is possible to keep four or five functional units going simultaneously in a hand coded particle mover.

3. Convenience

When a new code is being conceived and written, or when an old code is being improved, the rush to get the job done often allows us to do a poor job of planning, commenting and documenting the code. The result is a code that is inconvenient to run, particularly by the code's non-autors. Code maintenance may also be nearly impossible, even by the author. We quickly forget what we intended when we wrote the code, and if there are no comments imbedded in the code, we may be reduced to analyzing the Fortran to try to figure out what it does. If the code is well structured, this may not be difficult. If it is a mass of GO TO statements, it could be impossible. I wrote a data handling routine once for a particle code that had such convoluted logic that I could never understand the algorithm well enough to follow the logic of the code I had written. At one point, it passed all the tests, so I knew it worked, so I stopped working on it knowing that if it ever broke I wouldn't be able to fix it. By the time I finished the job, I knew a lot about structured programming and documentation.

The following paragraphs set out my recommendations on how to lay out and write a code for ease of use and maintenance and how to organize it so that it can be used conveniently without need for an intimate knowledge of how it works.

3.1 Overall Design

I always design my codes so that the graphical output is generated in a postprocessor. The main code saves diagnostic data periodically in as raw a form as possible. Decisions as to what data is to be taken and at what intervals and with what completeness and precision are made when the physics code is set up and run. The postprocessor reduces and processes the data and hen plots it graphically. Many decisions, particularly about graphics options can be made at postprocess time. Since the postprocessor can be rerun cheaply by comparison to the original physics run, lost fiche ore a mistyped graphical parameter is less likely to cause grief. The

other advantage of keeping the graphics routines in the postprocessor is that they cannot invade and overrun the physics code and thus destroy its legibility. The is no temptation to put graphics calls inside a particle mover, for instance.

The code is designed to be restartable. At predetermined intervals, a copy of all the working data is written to a set of files, with suitable names, at the end of the run, these files, along with the diagnostic files are written to permanent mass store. If it is desired to continue this run for say another hour, the data files are made available so that the code will not start from scratch, but resume where it left off. In addition to providing convenience of running the code, this design helps bullet proof the code against machine crashes.

The code is set up to look for data or commands to be entered from the keyboard during operation. Once each timestep, it looks for prearranged signals from the keyboard. It is therefore possible to stop the run, force a data dump, ask for data to be printed at the terminal, etc as the code runs. In principle, there is no limit to the degree of interactivity one could incorporate. It should be noted that the exact same code can be run in a batch subsystem without any need to interact with a terminal.

Parameter initialization is done using Namelist variables. All parameters are initialized to default values inside the code and only those that differ from the default values for a given run need be mentioned in the input deck. The input deck is either a file or it can be typed in at run time from the terminal if need be.

3.2 Postprocessor

The postprocessor is designed to produce graphical output from the data saved in the running of the physics code. The output can then be sent to a plotter of some kind or viewed directly on a graphics terminal. A more sophisticated postprocessor might allow one to design plots on the fly, to better examine some feature in the data for instance.

3.3 Internal Structure of the Code

Internally, the code should be organized into many, logically independent subroutines. No code block should be more than a few hundred lines in length, excluding comments, which should double its length. Liberal use of common blocks makes argument passing almost unnecessary. The idea is to make the code as easy to understand as possible.

In hand coded particle movers, I like to use four or five separate subroutines for the CAL mover, each to perform a logically separate part of the move. With each set of calls, a vector of 64 particles is moved. Within a Fortran bookkeeping subroutine, the sequence might be as follows: A CAL routine is called which unpacks the particles and calculates cell numbers and weights. Another CAL routine calculates the field values for the 64 particles. A third CAL routine advances the particles. A fourth CAL routine recalculates the cell numbers and weights and repacks the particles. Finally, the weights are used to accumulate the charge and current arrays in the last CAL routine. This is the basic cycle. Between these calls to the CAL mover, one can insert calls to Fortran diagnostic routines, particle injectors and routines to handle unusual boundary conditions. Thus it is possible to tightly intermix CAL and Fortran routines cleanly. Even at the level of moving

only 64 particles, the overhead of all these subroutine calls is small enough to be ignored.

In the previous section, I mentioned that particles can be packed and unpacked. By that I meant that it is often possible to compact the particle data by a factor of two in order to save space in memory or to reduce the time needed to transfer the data to and from disk. This is a common practice in codes such as this.